GPU acceleration of the matrix-free interior point method

Smith, E, Gondzio, J & Hall, J 2012, 'GPU acceleration of the matrix-free interior point method' Lecture Notes in Computer Science, vol 7203, pp. 681-689.

GPU Acceleration of the Matrix-Free Interior Point Method

Edmund Smith, Jacek Gondzio, and Julian Hall

School of Mathematics and Maxwell Institute for Mathematical Sciences,
University of Edinburgh, JCMB, King's Buildings, Edinburgh,
EH9 3JZ, United Kingdom
J.A.J.Hall@ed.ac.uk

Abstract. The *matrix-free* technique is an iterative approach to interior point methods (IPM), so named because both the solution procedure and the computation of an appropriate preconditioner require only the results of the operations $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$, where A is the matrix of constraint coefficients. This paper demonstrates its overwhelmingly superior performance on two classes of linear programming (LP) problems relative to both the simplex method and to IPM with equations solved directly. It is shown that the reliance of this technique on sparse matrix-vector operations enables further, significant performance gains from the use of a GPU, and from multi-core processors.

Keywords: interior point methods, linear programming, matrix-free methods, parallel sparse linear algebra.

1 Introduction

Since they first appeared in 1984 [9], interior point methods (IPM) have been a viable alternative to the simplex method as a means of solving linear programming (LP) problems [14]. The major computational cost of IPM is the direct solution of symmetric positive definite systems of linear equations. However, the limitations of direct methods for some classes of problems have led to iterative techniques being considered [1,3,11]. The *matrix-free* method of Gondzio [5] is one such approach and is so named because the iterative solution procedure and the computation of a suitable preconditioner require only the results of products between the matrix of constraint coefficients and a (full) vector. This paper demonstrates how the performance of the matrix-free IPM may be accelerated significantly using a Graphical Processing Unit (GPU) via techniques for sparse matrix-vector products that exploit common structural features of LP constraint matrices. To the best of our knowledge this is the first GPU-based implementation of an interior point method.

Section 2 presents an outline of the matrix-free IPM that is sufficient to motivate its linear algebra requirements. Results for two classes of LP problems demonstrate its overwhelmingly superior performance relative to the simplex method and to IPM with equations solved directly. Further analysis shows that

R. Wyrzykowski et al. (Eds.): PPAM 2011, Part I, LNCS 7203, pp. 681–689, 2012.

the computational cost of the matrix-free IPM on these problems is dominated by the iterative solution of linear systems of equations which, in turn, is dominated by the cost of matrix-vector products. Techniques for evaluating products with LP constraint matrices on multi-core CPU and many-core GPU are developed in Section 3. These techniques exploit commonly-occurring structural features of sparse LP constraint matrices. The results from an implementation with accelerated matrix-vector products show that significant speed-up in the overall solution time can be achieved for the LP problems considered, with the GPU implementation in particular providing large gains. Conclusions and suggestions for future work are offered in Section 4.

2 The Matrix-Free Interior Point Method

The theory of interior point methods [6,14] is founded on the following general primal-dual pair of linear programming (LP) problems.

$$\begin{array}{ll} \text{Primal} & \text{Dual} \\ \min \boldsymbol{c}^T\boldsymbol{x} & \max \boldsymbol{b}^T\boldsymbol{y} \\ \text{s. t. } A\boldsymbol{x} = \boldsymbol{b} & \text{s. t. } A^T\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c} \\ \quad \boldsymbol{x} \geq \boldsymbol{0} & \quad \boldsymbol{y} \text{ free}, \boldsymbol{s} \geq \boldsymbol{0}, \end{array} \tag{1}$$

where $A \in I\!R^{m\times n}$ has full row rank $m \leq n$, $\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{c} \in I\!R^n$ and $\boldsymbol{y}, \boldsymbol{b} \in I\!R^m$. IPMs employ logarithmic barrier functions to handle simple inequality constraints. The first order optimality conditions for the corresponding logarithmic barrier problems can be written as

$$\begin{aligned} A\boldsymbol{x} &= \boldsymbol{b} \\ A^T\boldsymbol{y} + \boldsymbol{s} &= \boldsymbol{c} \\ XS\boldsymbol{e} &= \mu\boldsymbol{e} \\ (\boldsymbol{x},\ \boldsymbol{s}) &\geq \boldsymbol{0} \end{aligned} \tag{2}$$

where X and S are diagonal matrices whose entries are the components of vectors $\boldsymbol{x}$ and $\boldsymbol{s}$ respectively and $\boldsymbol{e}$ is the vector of ones. The third equation $XS\boldsymbol{e} = \mu\boldsymbol{e}$ replaces the usual complementarity condition $XS\boldsymbol{e} = \boldsymbol{0}$ which holds at the optimal solution of (1). As μ is driven to zero in the course of a sequence of iterations, the vectors $\boldsymbol{x}$ and $\boldsymbol{s}$ partition into zero and nonzero components. In each IPM iteration, a search direction is computed by applying the Newton method to optimality conditions (2):

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I_n \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta x} \\ \boldsymbol{\Delta y} \\ \boldsymbol{\Delta s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\xi_p} \\ \boldsymbol{\xi_d} \\ \boldsymbol{\xi_\mu} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} - A\boldsymbol{x} \\ \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{s} \\ \mu\boldsymbol{e} - XS\boldsymbol{e} \end{bmatrix}. \tag{3}$$

By using the sets of equations in (3) to eliminate first $\boldsymbol{\Delta s}$, and then $\boldsymbol{\Delta x}$, the following symmetric positive definite **normal equations** system is obtained

$$(A\Theta A^T)\boldsymbol{\Delta y} = \boldsymbol{g}, \tag{4}$$

where $\Theta = XS^{-1}$ is a diagonal matrix. Since the normal equations matrix $A\Theta A^T$ is symmetric and positive definite, its LL^T Cholesky decomposition may be formed. In IPM, this is the usual means of solving directly for $\boldsymbol{\Delta y}$ and hence, by reversing the elimination process, $\boldsymbol{\Delta x}$ and $\boldsymbol{\Delta s}$. However, the density of $A\Theta A^T$ may be significantly higher than A, and the density of L may be higher still. For some large LP problems, the memory required to store L may be prohibitive. Following [6] test problems which exhibit this behaviour are given in Table 1. The first two problems are larger instances of quadratic assignment problems (QAP) [10] whose solution is one of the great challenges of combinatorial optimization. The remaining problems are part of a calculation from Quantum Physics of non-classicality thresholds for multiqubit states, and were provided to us by Jacek Gruca [7]. As problem size increases, the memory requirement of the Cholesky decomposition prevents them from being solved via standard IPM and the simplex method is seen not to be a viable alternative.

Table 1. Prohibitive cost of solving larger QAP problems and qubit problems using Cplex 11.0.1 IPM and dual simplex

	Dimensions			IPM		Simplex
Problem	Rows	Columns	Nonzeros	Cholesky Nonzeros	Time	Time
nug20	15,240	72,600	304,800	38×10^6	1034 s	79451 s
nug30	52,260	379,350	1,567,800	459×10^6	OoM	>28 days
1kx1k0	1,025	1,025	34,817	0.5×10^6	0.82 s	0.38 s
4kx4k0	4,097	4,097	270,337	8×10^6	89 s	11 s
16kx16k0	16,385	16,385	2,129,921	128×10^6	2351 s	924 s
64kx64k0	65,537	65,537	16,908,289	2048×10^6	OoM	111 h

For some LP problems the constraint matrix may not be known explicitly due to its size or the nature of the model, but it may nonetheless be possible to evaluate $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. Alternatively, for some problems there may be much more efficient means of obtaining these results than evaluating them as matrix-vector products. For such problems, Gondzio [5] is developing *matrix-free* IPM techniques in which systems of equations are solved by iterative methods using only the results of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. However, the present work is concerned with LPs for which which A is known explicitly but solution via standard IPM and the simplex method is impractical. This is the case for the problems given in Table 1.

Since the normal equations matrix $A\Theta A^T$ is symmetric and positive definite, the method of conjugate gradients can, in theory, be applied. However, its convergence rate depends on the ratio between the largest and smallest eigenvalues of $A\Theta A^T$, as well as the clustering of its eigenvalues [8]. Recall that since there will be many indices j for which only one of x_j and s_j goes to zero as the optimal solution is approached, there will be a very large range of values in Θ. This ill-conditioning means that conjugate gradients is unlikely to converge. Within

matrix-free IPM, the ill-conditioning of $A\Theta A^T$ is addressed in two ways: by modifying the standard IPM technique and by preconditioning the resulting normal equations coefficient matrix.

The optimization problem is regularized by adding quadratic terms $\boldsymbol{x}^T R_p \boldsymbol{x}$ and $\boldsymbol{y}^T R_d \boldsymbol{y}$ to the primal and dual objective in (1), respectively. Consequently, the matrix in the normal equations (4) is replaced by

$$G_R = A(\Theta^{-1} + R_p)^{-1} A^T + R_d, \tag{5}$$

in which R_p guarantees an upper bound on the largest eigenvalue of G_R and R_d guarantees that the spectrum of G_R is bounded away from zero. Therefore, for appropriate R_p and R_d the condition number of G_R is bounded regardless of the conditioning of Θ.

The convergence properties of the conjugate gradient method are improved by applying a preconditioner P which approximates the partial Cholesky decomposition of G_R

$$G_R = \begin{bmatrix} L_{11} \\ L_{21} \; I \end{bmatrix} \begin{bmatrix} D_L & \\ & S \end{bmatrix} \begin{bmatrix} L_{11}^T \; L_{21}^T \\ I \end{bmatrix} \approx P = \begin{bmatrix} L_{11} \\ L_{21} \; I \end{bmatrix} \begin{bmatrix} D_L & \\ & D_S \end{bmatrix} \begin{bmatrix} L_{11}^T \; L_{21}^T \\ I \end{bmatrix}. \tag{6}$$

Namely, in the preconditioner P, the Schur complement S is replaced with its diagonal D_S.

The number of nontrivial columns in the preconditioner is $k \ll m$ so, since only the diagonal entries of S are ever computed, the preconditioner is vastly cheaper to compute, store and apply than the complete Cholesky decomposition. Each iteration of the preconditioned conjugate gradient (PCG) method requires one operation with both P^{-1} and G_R. Since D_L, D_S, Θ, R_p and R_d are all diagonal matrices, the major computational costs are the operations with the nontrivial columns of P and the matrix-vector products with A and A^T. It is seen in Table 2 that the cost of PCG dominates the cost of solving the LP problem, and that PCG is dominated by the cost of operating with P^{-1} and calculating $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. For the QAP problems the cost of applying the preconditioner is significant, but for the quantum physics problems the cost of the

Table 2. Proportion of solution time accounted for by preconditioned conjugate gradients, operations with P^{-1} and calculations of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$

	Percentage of solution time			
Problem	PCG	P^{-1}	$A\boldsymbol{x}$	$A^T\boldsymbol{y}$
nug20	89	55	17	15
nug30	90	54	18	17
1kx1k0	62	41	12	11
4kx4k0	89	42	19	28
16kx16k0	87	30	30	29
64kx64k0	87	19	37	34

matrix-vector products dominates the solution time for the LP problem, and this effect increases for larger instances. Section 3 considers how the calculation of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$ may be accelerated by exploiting a many-core GPU and multi-core CPU.

3 Accelerating Sparse Matrix-Vector Products

The constraint matrix of a large LP problem is usually sparse and structured, and any competitive routine must take advantage of this fact. Sparse arithmetic presents different challenges as compared to dense arithmetic. Although sparse matrices of similar size to dense matrices can be multiplied much more quickly, it is in general the case, on both contemporary CPUs and GPUs, that the rate at which floating point operations are performed is lower; there are simply far fewer such operations.

We will consider the acceleration of the following operations

$$\boldsymbol{y} = A\boldsymbol{x} \quad (\text{FSAX}), \quad \boldsymbol{x} = A^T\boldsymbol{y} \quad (\text{FSATY}), \quad \boldsymbol{z} = (A\Theta A^T)\boldsymbol{y} \quad (\text{FSAAT}) \tag{7}$$

where $A \in \mathbb{R}^{m\times n}$ is sparse, $\Theta \in \mathbb{R}^{n\times n}$ is a diagonal matrix and, and the vectors $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$ and $\boldsymbol{z} \in \mathbb{R}^m$ are dense.

Memory bandwidth is a critical bottleneck for current generation CPUs. To alleviate this, high-speed cache is available to store recently used, or soon to be used, data. Caching is most beneficial when data items are re-used multiple times in quick succession. For operations requiring $O(n^3)$ operations on $O(n^2)$ data, such as matrix-matrix multiply, correct use of cache can give significant gains. In the case of matrix-vector multiply, there are only $O(n^2)$ operations: each data item is used once. Thus raw bandwidth determines performance on a large dataset, and caching may be expected to be mostly ineffective.

Vectorisation can give a significant performance gain for dense arithmetic on modern CPUs: identical operations are performed in parallel on a bank of data items (two at once for SSE2; four at once for AVX). Unfortunately, in sparse matrix-vector products, the elements of the vector corresponding to the non-zeroes in a given matrix row are widely separated in memory, and this makes efficient vectorisation difficult (in our tests, the packing costs outweighed the gains).

It has become typical for a CPU to contain multiple cores, independent execution units with some mutual dependencies, for example shared memory bandwidth and some shared cache. For a task like sparse matrix-vector multiply, which can be easily divided into a small number of independent pieces, there are no great problems with exploiting multiple cores. The effectiveness of the result though is less certain, given the essentially bandwidth limited nature of the problem in the first place. It should be noted that we shall be using a dual CPU system, so that double the bandwidth is available when all cores are used.

Current GPUs have large numbers of execution contexts, called threads, each of which is significantly slower and less able than a CPU core. A GPU benefits from potentially better memory performance and an explicitly managed cache, but that memory performance depends critically on achieving *coalesced*

accesses: adjacent threads must read adjacent memory locations. The challenge in mapping sparse matrix-vector products to a GPU, then, is predominantly in arranging for the task to be broken down into many small ones, and for those threads to access memory appropriately to preserve performance.

3.1 GPU Kernels

A number of kernels have been proposed in the literature for sparse matrix-vector products [2,4,12]. We summarise some of the key ideas below.

Threads on a GPU can only synchronise in a limited context - synchronisation within a warp is guaranteed, synchronisation within a block can be arranged. This means that any one element of the final result must be calculated by no more than a single block if the answer is to achieved without running the kernel multiple times. In the context of sparse matrix-vector products, this means each entry of the result vector may be calculated by at most one block.

If each thread calculates a row, then the data must be laid out so that the data for neighbouring rows are interleaved: this will mean adjacent threads read adjacent memory locations at each time step. This organisation has been called ELLPACK [2]. In practice, having each thread calculate a row under-utilises the device: insufficient parallelism is being identified.

An entire warp (thirty-two threads) can be used to calculate a row by first performing all the multiplications and storing the result in cache (shared memory), then performing a parallel reduce. A warp is automatically synchronised so there is no synchronisation overhead in this algorithm. If this is done with variable run-length storage of the rows, the result is vector CSR [2].

When the lengths of the rows vary considerably, it can be a problem both for load balancing, and for memory requirements. Pure ELLPACK is not practical for matrices with dense rows. ELLR-T [12] can eliminate some of this inefficiency by storing the length of each row, but the need to reserve enough memory for the matrix to be stored as fully dense remains. The HYB [2] kernel overcomes this limitation by storing a core of the problem as ELL, and any extra elements in long rows as COO (unstructured, sparse). Unfortunately, COO is not an especially fast kernel.

If the constraint matrix has dense blocks which can be identified, data blocking can give significant speedup [4].

Our target problems have rows and columns of mostly identical length, barring a fully dense row and a fully dense column. They do not lend themselves to data blocking. The kernels discussed below were optimized for these problems.

We considered three families of kernel. Firstly, dense-hybrid ELL (DHELL) in which dense rows are extracted for treatment by a block of their own, and the remainder are stored in ELL format. Secondly, vector CSR as discussed by [2]. Finally, dense-hybrid transpose ELL (DHTELL), in which the matrix of indices and coefficients is transposed relative to that encountered in ELL. It can be seen also as CSR with a fixed row length. This kernel is novel.

As for [12], we considered using different numbers of threads per row. The ELL format (equivalently, ELLR-T) hampers such explorations, because an entire

half-warp must read adjacent memory locations. Thus the maximum number of threads per row is the same as the number of half-warps per block, usually sixteen. For both CSR and TELL formats, an entire block can be brought to bear on a row (256 threads), but the minimum number of threads per row falls to sixteen.

The best performing kernel was DHTELL with a half-warp per row (sixteen threads) and a block devoted to any dense part. Although this level of parallelism could be matched in the DHELL kernel, the former was marginally faster. There is little to choose between any of the vectorised kernels, when compared to the basic CSR or ELL kernels.

Note that the constraint matrix is stored twice on the GPU, once row-wise and once column-wise, to allow operations with the transpose. Any naïve implementation of the alternative would require impractical amounts of memory in which to accumulate partial results.

The only significant optimization for this platform which has not been considered, that we are aware of, is use of the texture cache to store the input vector. Results presented for band diagonal matrices in [2] suggest this as a possible future enhancement.

3.2 Results

The following results are obtained from a test system having two AMD Opteron 2378 (Shanghai) quad-core processors, 16 GiB of RAM and a Tesla C2070 GPU with 6 GiB of RAM. Note that the processors are relatively slow in serial, though the NUMA configuration of the memory bus gives high parallel memory performance. The GPU is a significantly more highly powered unit, making raw speed characterisations of less interest than the potential for improvement with a given investment.

Table 3. Comparison of accelerated matrix-free IPM codes. All times include data transfer.

	Solve time (s)			SpMV time (s)		
Problem	Serial	8 core	GPU	Serial	8 core	GPU
nug20	2.19	1.18	1.60	1.49	0.495	0.945
nug30	20.5	15.8	15.4	15.1	9.69	9.45
1kx1k0	0.244	0.177	0.217	0.0360	0.0128	0.0506
4kx4k0	3.03	2.06	2.15	1.06	0.218	0.336
16kx16k0	24.9	18.4	13.5	13.1	6.70	1.72
64kx64k0	170.0	109.0	74.0	115.0	47.4	12.2
96kx128-0	137.0	71.1	58.8	93.4	28.6	15.3
256x256-0	866.0	283.0	222.0	699.0	119.0	56.4

Speed-up of sparse matrix-vector kernels using all eight cores of the test system is between two and six times, giving at most a threefold speed-up of the IPM solution time. Using the high powered GPU, speed-up of these same kernels can approach ten times, though overall solution time is reduced by no more than a factor of four. Clearly significant speed-up of matrix-free interior point, whether by many-core or multi-core parallelism, is possible.

4 Conclusions

The matrix-free approach shows promise in making some of the most difficult classes of problem tractable by interior point methods. Its focus on a small core of sparse operations makes highly optimized implementations using state of the art hardware possible without excessive difficulty.

The particular choice of many-core or multi-core acceleration depends on the hardware available. As has been noted elsewhere [13], a GPU can provide performance essentially equivalent to a small number of multi-core processors in the context of sparse problems.

References

1. Al-Jeiroudi, G., Gondzio, J., Hall, J.: Preconditioning indefinite systems in interior point methods for large scale linear optimization. Optimization Methods and Software 23(3), 345–363 (2008)
2. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. Tech. Rep. NVR-2008-004, NVIDIA Corporation (2008)
3. Bergamaschi, L., Gondzio, J., Zilli, G.: Preconditioning indefinite systems in interior point methods for optimization. Computational Optimization and Applications 28, 149–171 (2004)
4. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 115–126. ACM (2010)
5. Gondzio, J.: Matrix-free interior point method. Computational Optimization and Applications, published online October 14 (2010), doi:10.1007/s10589-010-9361-3
6. Gondzio, J.: Interior point methods 25 years later. European Journal of Operational Research, published online October 8 (2011), doi:10.1016/j.ejor.2011.09.017
7. Gruca, J., Wiesław, L., Żukowski, M., Kiesel, N., Wieczorek, W., Schmid, C., Weinfurter, H.: Nonclassicality thresholds for multiqubit states: Numerical analysis. Physical Review A 82 (2010)
8. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Natl. Bur. Stand 49, 409–436 (1952)
9. Karmarkar, N.K.: A new polynomial–time algorithm for linear programming. Combinatorica 4, 373–395 (1984)
10. Nugent, C.E., Vollmann, T.E., Ruml, J.: An experimental comparison of techniques for the assignment of facilities to locations. Operations Research 16, 150–173 (1968)
11. Oliveira, A.R.L., Sorensen, D.C.: A new class of preconditioners for large-scale linear systems from interior point methods for linear programming. Linear Algebra and its Applications 394, 1–24 (2005)

12. Vázquez, F., Ortega, G., Fernández, J., Garzón, E.: Improving the performance of the sparse matrix vector product with GPUs. In: 2010 10th IEEE Conference on Computer and Information Technology (CIT 2010), pp. 1146–1151 (2010)
13. Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., Shringapure, A.: On the limits of GPU acceleration. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism. USENIX Association (2010)
14. Wright, S.J.: Primal-Dual Interior-Point Methods. SIAM, Philadelphia (1997)